Announcements

- **Project 1** – phase A due Wed Jan 16\textsuperscript{th} 11pm via catalyst
  – Be sure to look at clarifications page!
- **Homework 1** – due Friday Jan 18\textsuperscript{th} at beginning of class
  – Clarifications posted
- **Homework 2** – due Friday Jan 25\textsuperscript{th} – coming soon!
- No class on Monday Jan 21\textsuperscript{st}

Today

- Priority Queues
- Binary Min Heap implementation

Review

- Priority Queue ADT: \texttt{insert} comparable object, \texttt{deleteMin}
- Binary heap data structure: Complete binary tree where each node has priority value greater than its parent
- \(O(\text{height-of-tree}) = O(\log n)\) \texttt{insert} and \texttt{deleteMin} operations
  - \texttt{insert}: put at new last position in tree and percolate-up
  - \texttt{deleteMin}: remove root, put last element at root and percolate-down
- But: tracking the “last position” is painful and we can do better

Array Representation of Binary Trees

From node 1:
- left child: \(1*2\) + 1
- right child: \(1*2+1\)
- parent: \(1/2\)

(wasting index 0 is convenient for the index arithmetic)

http://xkcd.com/163
Pseudocode: insert

```c
void insert(int val) {
    if(size==arr.length-1) resize();
    size++;
i = percolateUp(size, val);
    arr[i] = val;
}
```

Percolate Up:

```c
while (hole > 1 && val < arr[hole/2]) {
    arr[hole] = arr[hole/2];
    hole = hole / 2;
}
return hole;
```

Example: After insertion

1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin

Example: After deletion

1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin

Other operations:

- **decreaseKey:** given pointer to object in priority queue (e.g., its array index), lower its priority value by \( p \)
  - Change priority and percolate up

- **increaseKey:** given pointer to object in priority queue (e.g., its array index), raise its priority value by \( p \)
  - Change priority and percolate down

- **remove:** given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - `decreaseKey` with \( p = \infty \) then `deleteMin`

Running time for all these operations?
Evaluating the Array Implementation…

Advantages:

- **Minimal amount of wasted space:**
  - Only index 0 and any unused space on right in the array
  - No "holes" due to complete tree property
  - No wasted space representing tree edges

- **Fast lookups:**
  - Benefit of array lookup speed
  - Multiplying and dividing by 2 is extremely fast (can be done through bit shifting (see CSE 351)
  - Last used position is easily found by using the PQArray's size for the index

Disadvantages:

- What if the array gets too full (or wastes space by being too empty)? Array will have to be resized.

Advantages outweigh Disadvantages: This is how it is done!

---

So why O(1) average-case insert?

- Yes, insert's **worst case** is O(log n)
- The trick is that it all depends on the order the items are inserted (What is the worst case order?)
- Experimental studies of randomly ordered inputs shows the following:
  - Average 2.807 comparisons per insert (# of percolation passes)
  - An element usually moves up 1.607 levels
- **deleteMin** is average O(log n)
  - Moving a leaf to the root usually requires re-percolating that value back to the bottom

---

Aside: Insert run-time: Take 2

- Insert: Place in next spot, percolUp
- How high do we expect it to go?
- Aside: Complete Binary Tree
  - Each full row has 2x nodes of parent row
  - 1+2+4+8+...+2^h = 2^(h+1) - 1
  - Bottom level has ~1/2 of all nodes
  - Second to bottom has ~1/4 of all nodes
- PercolUp Intuition:
  - Move up if value is less than parent
  - Inserting a random value, likely to have value not near highest, nor lowest; somewhere in middle
  - Given a random distribution of values in the heap, bottom row should have the upper half of values, 2^h from bottom row, next 1/4 - Expect to only raise a level or 2, even if h is large
- Worst case: still O(log n)
- Expected case: O(1)
- Of course, there’s no guarantee; it may percolUp to the root

---

Building a Heap

Suppose you have n items you want to put in a new priority queue

- A sequence of n insert operations works
- Runtime?

Can we do better?

- If we only have access to insert and deleteMin operations, then NO.
- There is a faster way - O(n), but that requires the ADT to have a specialized buildHeap operation

Important issue in ADT design: how many specialized operations?

- Tradeoff: Convenience, Efficiency, Simplicity

---

Floyd’s buildHeap Method

Recall our general strategy for working with the heap:

- Preserve structure property
- Break and restore heap property

Floyd’s buildHeap:

1. Create a complete tree by putting the n items in array indices 1,...,n
2. Treat the array as a heap and fix the heap-order property
   - Exactly how we do this is where we gain efficiency

---

Floyd’s buildHeap Method

Bottom-up:

- Leaves are already in heap order
- Work up toward the root one level at a time

```java
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val = arr[i];
    hole = percolateDown(i, val);
    arr[hole] = val;
  }
}
```
**buildHeap Example**

- Say we start with this array:
  \[12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6\]
- In tree form for readability
  - Red for node not less than descendants
- heap-order problem
  - Notice no leaves are red
  - Check/fix each non-leaf bottom-up (6 steps here)

1/16/2013

**Step 1**

- Happens to already be less than child

1/16/2013

**Step 2**

- Percolate down (notice that moves 1 up)

1/16/2013

**Step 3**

- Another nothing-to-do step

1/16/2013

**Step 4**

- Percolate down as necessary (steps 4a and 4b)

1/16/2013

**Step 5**

- Another nothing-to-do step

1/16/2013
**buildHeap Example**

```
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
```

**But is it right?**

- “Seems to work”
  - Let’s prove it restores the heap property (correctness)
  - Then let’s prove its running time (efficiency)

```c
void buildHeap() {
    for (i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i,val);
        arr[hole] = val;
    }
}
```

**Correctness**

```
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i,val);
        arr[hole] = val;
    }
}
```

Loop Invariant: For all $j > i$, $arr[j]$ is less than its children
- True initially: If $j > size/2$, then $j$ is a leaf
  - Otherwise its left child would be at position $> size$
- True after one more iteration: loop body and `percolateDown` make $arr[i]$ less than children without breaking the property for any descendants
So after the loop finishes, all nodes are less than their children

**Efficiency**

```
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i,val);
        arr[hole] = val;
    }
}
```

Easy argument: `buildHeap` is $O(n \log n)$ where $n$ is `size`
- `size/2` loop iterations
- Each iteration does at most $O(\log n)$

This is correct, but there is a more precise (“tighter”) analysis of the algorithm...

**Lessons from buildHeap**

- Without `buildHeap`, our ADT already let clients implement their own in $\Theta(n \log n)$ worst case
  - Worst case is inserting lower priority values later
- By providing a specialized operation internally (with access to the data structure), we can do $O(n)$ worst case
  - Intuition: Most data is near a leaf, so better to percolate down
- Can analyze this algorithm for:
  - Correctness: Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First analysis easily proved it was $O(n \log n)$
    - A “tighter” analysis shows same algorithm is $O(n)$
What we’re skipping (see text if curious)

- **d-heaps**: have \( d \) children instead of 2 (Weiss 6.5)
  - Makes heaps shallower, useful for heaps too big for memory
  - How does this affect the asymptotic run-time (for small \( d \)'s)?
- **Leftist heaps, skew heaps, binomial queues** (Weiss 6.6-6.8)
  - Different data structures for priority queues that support a logarithmic time merge operation (impossible with binary heaps)
  - **merge**: given two priority queues, make one priority queue
  - Insert & deleteMin defined in terms of merge

Aside: How might you merge binary heaps:

- If one heap is much smaller than the other?
- If both are about the same size?

```c
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i, val);
        arr[hole] = val;
    }
}
```